A Comparison of Ada and C on Sun and microVAX

Intelligent Controls Group
Robot Systems Division

Principle Author: Stephen Leake
Date: February 16, 1988

Document number: ICG-#8
Document version: 1.2
Document approval: _____

Revisions

1. Stephen Leake
   20 June 1988
   Add timing for Verdix version 1.5.
2. Stephen Leake
   4 August 1988
   Correct minor errors

Scope of the Document

This document describes the results of work that compares the suitability of the Ada and C programming languages for the NASREM robot control system, on VAX and Sun hardware. Execution speed, portability, and abstraction are considered for each compiler / machine combination.

A Comparison of Robot Kinematics in Ada and C on Sun and microVAX

Stephen Leake
Intelligent Controls Group
Robot Systems Division
National Bureau of Standards

There are many considerations involved in choosing a language for any large and long-lived project. This study addresses just three issues; execution speed, portability to different computing hardware, and level of abstraction. Robot kinematics was chosen as the example application, because it is typical of computations in robotics, and because the algorithms are fairly well known. The forward and inverse kinematics were implemented on five compiler/machine pairs: DEC Ada on a μVAX II, DEC C on a μVAX II, Verdix Ada on a Sun 3/160, Sun C on a Sun 3/160, and SMACRO (a macro assembler) on 8086/8087. In summary, Ada was judged better for abstraction, and neither Ada nor C had any particular advantage for portability. On the VAX, DEC Ada is faster than DEC C, while on the Sun , Sun C is faster than Verdix Ada. Sun C is slightly faster than DEC Ada, but all execution times, including SMACRO on the 8086/8087, are within a factor of 2. The Ada implementation was done in two ways; first using a more abstract method, second using a less abstract method. The level of abstraction had a significant impact on execution speed, as did the various Ada run time checks.

1. Introduction

There are many considerations involved in choosing a language for a large and long-lived project such as the Flight Telerobot Servicer (FTS) [Purves 87]. There will be multiple programmers, both for the initial effort and during maintenance. The project involves real-time control of mechanical systems. There are many safety and reliability issues. The system can be expected to evolve over time, both in functionality and in the computing hardware it runs on.

This study addresses just three issues; execution speed, portability to different computing hardware, and level of abstraction. Execution speed is critical in real-time control. Portability is necessary both for future growth, and for technology transfer to the commercial market. Level of abstraction may be defined in terms of the object oriented programming philosophy [Booch 86]. This holds that the structure of the code should be as close to the human view of the task as possible. The closer the code is to this goal, the more abstract it is.

As is well known, execution speed for any particular computer is highly application dependent. In this study, the forward and inverse kinematics for the PUMA 760 robot are used as the test application. This application is fairly typical of computation tasks in the PRIM and SERVO levels of NASREM, and to some extent in the World Model [NASREM 87]. It is not representative of computation tasks in the other parts of NASREM, such as path planning or vision processing. The inverse kinematics is coded in a relatively low-level way, for speed. The forward kinematics is coded in a more high-level way, for two reasons. First, this serves as an example of high-level prototype code. Typically, we are not as concerned with speed when prototyping; getting the code to work correctly is more important. Secondly, the high-level implementation lets us study the trade-off involved between level of abstraction and speed.

Execution speed is also compiler and machine dependent. It is virtually impossible to separate the compiler from the machine; the back end of a compiler is highly machine dependent. This study involves five compiler/machine pairs: DEC Ada version 1.4 on a μVAX II, DEC C version 2.3 on a μVAX II, Verdix Ada versions 5.41 and 5.5h on a Sun 3/160, Sun C version 3.2 on a Sun 3/160, and SMACRO on 8086/8087. (SMACRO is the NBS-developed language used to write RCS (Real-time Control System), the precursor to NASREM). The SMACRO code is included only in the execution timing study, as an example of what assembler level manual optimization can achieve.

The same programmer (the author) wrote all of the application code, assuring that the programming style and algorithms are as close as reasonable. This is not to say that they are identical; each language encourages its own style. At the most abstract level, the algorithms are identical (and they produce the same numerical results), but at the lowest level, the number and order of multiplies and adds is slightly different among the three languages, and probably different between the two implementations of Ada and C, due to optimization differences.

The rest of the paper is organized as follows. Section 2 presents the application in more detail, describing the algorithms used. Section 3 presents the Ada and C implementations, and discusses the level of abstraction, portability, and execution speed for each implementation. Section 4 summarizes the conclusions.

Commercial equipment is identified in this paper in order to adequately describe the systems under test. In no case does such identification imply recommendation by the National Bureau of Standards, nor does it imply that this equipment was necessarily the best for the purpose.

This research has been supported by NASA under Contract S-28176-D.

2. Application description

This section describes the application, and the algorithms used.

In any robot control system, the robot kinematics plays an important role. It translates between Cartesian space, where tasks are planned, and joint space, where the robot actuators are. In a typical application, a trajectory is generated by producing Cartesian points at a regular rate (20 to 100 milliseconds), and the inverse kinematics converts the Cartesian points to joint values at the same rate. The forward kinematics may be used to convert sensed joint positions to Cartesian positions, usually at the same rate as the inverse kinematics.

There are many derivations of the kinematics for the PUMA 760 and other robots [Craig 86], [Paul 81]. The algorithm used here for the inverse kinematics is similar, and will not be derived in detail. The algorithm for the forward kinematics, however, is significantly different.

The typical approach to deriving the forward kinematics is as follows:

1) Define Cartesian frames and joint origins for each link of the robot.

2) Define the transformation matrices for each joint.

3) Multiply out the matrices, and simplify as much as possible.

My approach is the same for steps 1 and 2, but I do not do step 3. The expression for the forward kinematics obtained from step 2 is:

$$^0T_6 = {}^0T_1\ {}^1T_2\ {}^2T_3\ {}^3R_4\ {}^4R_5\ {}^5R_6,$$

where

$$^0T_1 = {}^0R_1\ {}^0P_1$$

$$^1T_2 = {}^1R_2\ {}^1P_2$$

$$^2T_3 = {}^2R_3\ {}^2P_3,$$

and ^iR_j and ^iP_j represent pure rotations and pure translations, respectively, from frame i to frame j, and T represents mixed rotation and translation.

Instead of trying to simplify the resulting expression, I code the indicated transform multiplies. If each transform is expressed as a homogeneous matrix, there are many zeroes, which makes the transform multiply very inefficient. However, there are more efficient representations of transforms. In the application code, I implemented four different

left	right	result	*	±
Matrix	Matrix	Matrix	27	18
Matrix	Angle_Frame_Axis	Matrix	12	6
Matrix	Cartesian	Cartesian	9	6
Matrix	Distance_Frame_Axis	Cartesian	3	0
Quaternion	Quaternion	Quaternion	16	12
Quaternion	Angle_Frame_Axis	Quaternion	8	4
Quaternion	Cartesian	Cartesian	39	21
Quaternion	Distance_Frame_Axis	Cartesian	12	5
Trig_Frame_Axis	Cartesian	Cartesian	4	2
Trig_Frame_Axis	Distance_Frame_Axis	Cartesian	2	0

Table 1. Operations count for various representations

representations of rotation, and two of translation, together with the associated multiply operations.

The four rotation representations are:

1) Trig_Frame_Axis: The sine and cosine of the signed magnitude of rotation, and an enumeral giving one of the principal axes (X, Y, Z) of a Cartesian frame. The rotation is about the indicated axis.

2) Angle_Frame_Axis: An angle giving the signed magnitude of rotation, and an enumeral, as in Trig_Frame_Axis.

3) Quaternion: Four numbers, given by [Cos ($\theta/2$), **n** Sin ($\theta/2$)], where θ is the magnitude of rotation, and **n** is the axis. Quaternions where first described by Hamilton [Hamilton 1844], and are also known as Euler parameters [Kane 83].

4) Matrices: The usual 3 by 3 orthonormal matrices.

The two representations of translations are:

1) Cartesian: The usual 3 element Cartesian vector.

2) Distance_Frame_Axis: The signed magnitude of the translation, and an enumeral as in Trig_Frame_Axis. The translation is along the indicated axis.

Note that the traditional homogeneous transformation is just a combination of an orthonormal matrix and a Cartesian vector. Obviously, the two rotation representations involving frame axes cannot represent arbitrary rotations. However, they do represent an important special case. All six of the rotations involved in the PUMA forward kinematics are about one of the axes of the previous link's frame. This is why there are so many zeroes in the matrix representation. Similarly, the translations are also along the frame axes.

The multiply operations for representations based on the frame axes are much simpler than for the general case. A summary of the number of operations involved is given in Table 1. It

is clear that the number of operations can be significantly reduced by using the appropriate representation. However, they will not be reduced below the number achieved by the traditional approach, and in fact, my approach usually keeps a few extra. Thus for pure speed, the traditional approach, involving manual simplification, is better.

However, we are concerned with more than pure speed: abstraction and maintenance play a role here. Using the frame_axis representations, the code is more abstract; closer to the user's view of the task. When we think of the link transforms for the PUMA robot, we visualize rotations about the axes of the frames established for each link, not the corresponding matrices or quaternions. The degree to which this is important depends on the language, as we shall discuss in the next section.

Debugging is certainly simpler; the programmer codes only the six transform multiplies, not the complex equations resulting from manual simplification. (Of course, the primitive multiply operations are assumed to be previously debugged). This is more important in prototype tasks; if a task is to be repeated as often as the kinematics are, the debugging time is less important than speed. However, there will be many prototype tasks in a growing, experimental system such as the FTS. Even for time-critical tasks, the algorithm can be proven by writing abstract code, then speeded up by replacing the abstract portions with low-level code. Then a known working version is available to help debug the low-level version.

Maintenance is enhanced, primarily because the code is more abstract, and therefore easier to understand and modify. The code is also more compact; there are fewer statements, and therefore fewer opportunities for error during maintenance.

In summary, the test application consists of two procedures; the inverse kinematics, coded in a conventional style, and the forward kinematics, coded in a more abstract, high-level style. These two procedures are somewhat typical of the types of procedures encountered in the PRIM and SERVO levels of NASREM. Keep in mind that the forward kinematics could be more highly optimized, but that it is serving as an example of a prototype procedure, where correctness and abstractness is more important.

3. Language evaluation

Figures 1 thru 4 show the Ada and C code for the forward and inverse kinematics routines (only part of the inverse kinematics routine is shown, due to its length). The SMACRO code is not shown. The following subsections address the issues of abstraction, portability, and execution speed. There are no measures for abstraction; a discussion of the issues is the best we can do, while recognizing that the discussion is subjective. Portability is somewhat more amenable to measurement; the problems can be documented in terms of compiler error messages and/or run-time bugs. A completely portable application would compile and run with no bugs, and no editing of any source code (this is, of course, almost never achieved in practice). If there are bugs or compiler errors, there is still a question of how "important" the problem is. Execution speed is of course fairly easy to measure.

```
procedure Forward_Kinematics (
    Geometry     : in GEOMETRY_RECORD;
    Joint        : in JOINT_ARRAY;
    Rot_Rep      : in Rotations.ROT_REPS;
    Pose         : in out Poses.POSE;
    Flags        : out CONFIG_FLAGS)
is
    -- compute the Cartesian pose and configuration flags, given the joint
    -- angles.

    use Translations, Rotations, Poses;

    Trig_0 : TRIG_PAIR := Sin_Cos (Joint (0)); -- used in computing flags

begin
    case Rot_Rep is     -- choose rotation representation
    when MATRIX =>
        Pose := (To_Matrix (ROTATION'(TRIG_FRAME_AXIS, (Trig_0, Z))),
            Ident_Translation)

    when others =>
        Pose := (To_Quaternion(ROTATION'(TRIG_FRAME_AXIS, (Trig_0, Z))),
            Ident_Translation)
    end case;

    Pose := Pose
        * TRANSLATION'(DISTANCE_FRAME_AXIS, (Geometry.Shoulder, Y))
        * ROTATION'(ANGLE_FRAME_AXIS, (Joint (1), Y))
        * TRANSLATION'(DISTANCE_FRAME_AXIS, (Geometry.Upper_Arm, Z))
        * ROTATION'(ANGLE_FRAME_AXIS, (Joint (2), Y))
        * TRANSLATION'(DISTANCE_FRAME_AXIS, (Geometry.Fore_Arm, Z))
        * ROTATION'(ANGLE_FRAME_AXIS, (Joint (3), Z))
        * ROTATION'(ANGLE_FRAME_AXIS, (Joint (4), Y))
        * ROTATION'(ANGLE_FRAME_AXIS, (Joint (5), Z));

    -- compute flags
    ...
end Forward_Kinematics
```

Figure 1. Ada code for forward kinematics.

```
POSE_FLAGS  forward_kinematics (geometry, in_joints)
GEOMETRY    *geometry;
JOINT_ARRAY in_joints;
    /* compute the Cartesian pose and configuration flags, */
    /* given the joint angles.*/
{
    auto    POSE_FLAGS  result;
    auto    TRIG_PAIR   trig_0; /* used in computing flags */

    trig_0 = sin_cosd (in_joints [0]);

    result.pose = pose_times_trig_frame_axis (ident_pose,
        trig_0, Z);

    result.pose = pose_plus_distance_frame_axis (result.pose,
        geometry->shoulder, Y);

    result.pose = pose_times_angle_frame_axis (result.pose,
        in_joints [1], Y);

    result.pose = pose_plus_distance_frame_axis (result.pose,
        geometry->upper_arm, Z);

    result.pose = pose_times_angle_frame_axis (result.pose,
        in_joints [2], Y);

    result.pose = pose_plus_distance_frame_axis (result.pose,
        geometry->fore_arm, Z);

    result.pose = pose_times_angle_frame_axis (result.pose,
        in_joints [3], Z);

    result.pose = pose_times_angle_frame_axis (result.pose,
        in_joints [4], Y);

    result.pose = pose_times_angle_frame_axis (result.pose,
        in_joints [5], Z);

    /* find configuration flags */
    ...
}
```

Figure 2. Sun C code for forward kinematics.

```
procedure Inverse_Kinematics
    (Geometry: in GEOMETRY_RECORD;
     Pose     : in Poses.POSE;
     Flags    : in CONFIG_FLAGS;
     Old_Joint: in JOINT_ARRAY;
     Joint    : in out JOINT_ARRAY)
is
    use Translations.Cartesians, Rotations.Quaternions;

    Rot_0_6 : Quaternion :=
         Rotations.To_Quaternion (Pose.Rotation).Quaternion;
        -- rotation part of T06.

    Tran_0_6 : Cartesian :=
         Translations.To_Cartesian (Pose.Translation).Cartesian;
        -- translation part of T06.

    X_1_6 : METERS; -- x component of T16.
    Trig_0, Trig_1,Trig_2 : TRIG_PAIR;
    R_3_6 : QUATERNION;  -- rotation component of T36.

begin
    JOINT_0_SOLUTION:
    declare
        Denom : METERS_2 := Tran_0_6 (X) ** 2 + Tran_0_6 (Y) ** 2;
    begin
        if Flags.Shoulder = RIGHTY
        then
              X_1_6 := - Sqrt (Tran_0_6 (X) ** 2 + Tran_0_6 (Y) ** 2
                   - Geometry.Shoulder ** 2);
        else
              X_1_6 :=   Sqrt (Tran_0_6 (X) ** 2 + Tran_0_6 (Y) ** 2
                   - Geometry.Shoulder ** 2);
        end if;

        Trig_0 := Make_Trig_Pair ((Tran_0_6 (Y) * X_1_6
                   - Geometry.Shoulder * Tran_0_6 (X)) / Denom,
              (Tran_0_6 (X) * X_1_6
                   + Geometry.Shoulder * Tran_0_6 (Y)) / Denom);

        Joint (0) := Atan2 (Trig_0);

    exception
        when NEGATIVE_SQUARE_ROOT => raise INNER_REACH_LIMIT;
    end JOINT_0_SOLUTION;

    -- other joints
    ...
end Inverse_Kinematics;
```

Figure 3. Ada code for inverse kinematics, header and Joint 0..

```
extern JOINT_STATUS inverse_kinematics (geometry, in_pose_flags,
    old_joints, new_joints)
GEOMETRY      *geometry;
POSE_FLAGS    in_pose_flags;
JOINT_ARRAY   old_joints;
JOINT_ARRAY   new_joints;
{
    auto    double            wrist_singularity_delta = 2.0E-2;
    auto    QUATERNION        rot_0_6;     /* rotation part of T06 */
    auto    CARTESIAN         tran_0_6;    /* translation part of T06 */
    auto    double            x_1_6;       /* x component of T16 */
    auto    TRIG_PAIR         trig_0, trig_1, trig_2;
    auto    JOINT_ARRAY       temp_joints;
    auto    QUATERNION        r_3_6;       /* rotation component of T36 */

    rot_0_6.s = in_pose_flags.pose.quaternion.s;
    rot_0_6.x = in_pose_flags.pose.quaternion.x;
    rot_0_6.y = in_pose_flags.pose.quaternion.y;
    rot_0_6.z = in_pose_flags.pose.quaternion.z;
    tran_0_6.x = in_pose_flags.pose.cartesian.x;
    tran_0_6.y = in_pose_flags.pose.cartesian.y;
    tran_0_6.z = in_pose_flags.pose.cartesian.z;

    {   /* joint_0_solution */
        auto      double      denom_1 = sqr (tran_0_6.x) + sqr (tran_0_6.y);
        auto      double      temp;

        /* check sign before calling sqrt */
        temp = sqr (tran_0_6.x) + sqr (tran_0_6.y) - sqr (geometry->shoulder);

        if (temp < 0.0) return INNER_REACH_LIMIT;

        if (in_pose_flags.flags.shoulder == RIGHTY)
            x_1_6 = - sm_sqrt (temp);
        else
            x_1_6 =   sm_sqrt (sqr (tran_0_6.x) + sqr (tran_0_6.y)
                - sqr (geometry->shoulder));

        trig_0 = make_trig_pair ((tran_0_6.y * x_1_6
            - geometry->shoulder * tran_0_6.x) / denom_1,
            (tran_0_6.x * x_1_6
                + geometry->shoulder * tran_0_6.y) / denom_1);

        temp_joints [0] = atand2 (trig_0);
    }
    /* other joints */
    ...
}
```

Figure 4. Sun C code for inverse kinematics, header and joint 0.

3.1. Abstraction

There are some fairly obvious differences between the Ada and C versions. One is that Ada supports aggregates, and C does not, except for the initialization of static variables (DEC C allows the initialization of automatic aggregate variables, Sun C does not). This makes the code more abstract: the contents of an aggregate variable are treated as a unit, rather than as discrete components. It makes the code more readable, and less prone to minor typing errors.

In the forward solution, the Ada version has a single assignment, with eight transforms multiplied together. This is exactly the same as the abstract expression for ${}^{0}T_{6}$. The C code is similar, but it involves an explicit temporary variable, and function names other than "*". The function names are rather long; obviously, abbreviations could be used. (They were not here for clarity).

Note that the arguments to the Ada functions are aggregates, while the inputs to the C functions are not. C does not have a construct similar to the Ada aggregate, as noted above. Aggregates allow the Ada code to be more flexible; the inputs to the multiply functions can be variables or aggregates; the inputs to the C functions can have only the form shown. A different C function must be called if we wish to multiply a pose by a variable containing an angle_frame_axis. For example, we would need:

function pose_times_angle_frame_axis_var (pose, angle_frame_axis)

function pose_times_angle_frame_axis_agg (pose, angle, frame_axis)

This is a good example of abstraction; an angle_frame_axis is an abstract entity, and in Ada it is handled in the same way whether it is a named variable or an aggregate of components, but in C it is not.

Another difference in the forward solution is in the choice of the rotation representation. The C code always uses quaternions as intermediates, and a pose is always expressed by a quaternion rotation and a Cartesian translation. The input to the Ada "*" function is a record with variant parts, one variant for each rotation (and translation) representation. Thus the same function handles all representations. This is why the case statement at the beginning of the forward kinematics can determine the final representation; the "*" function uses matrices if either of the inputs is a matrix, and quaternions otherwise. This is another example of abstraction; some knowledge of which representation is best is encoded in the "*" routine itself, and thus is transparent to the user. If a particular routine is passed a matrix, it will use that representation, and if it is passed a quaternion, it will use that. It would be possible to do similar things in C, using unions. It would be less abstract, since C does not support operator overloading and aggregates. In addition, the strong typing in Ada makes debugging such code much simpler; debugging union structures in C is difficult.

One final difference is the use of exceptions in Ada. In the inverse kinematics, if the input pose is too close to the base, the robot cannot reach it. This shows up in the algorithm as a square root with a negative argument. In Ada, we can let the exception mechanism deal with this special case; here we convert the exception NEGATIVE_SQUARE_ROOT (defined in my math package) to the exception INNER_REACH_LIMIT. In C, using exceptions makes the high-level code operating system dependent, since the language does not define a

standard interface to exceptions (see section 3.2.1 below). Instead, we explicitly check for a negative argument, and return a status indicating the problem. The Ada code is more abstract; it indicates that the negative square root is a special case, and, since it propagates an exception, it forces the calling routine to handle the problem differently than a normal return. In C, the calling routine must explicitly check the status, thus making all cases appear of equal importance. Indeed, the calling routine can choose to ignore the status, resulting in erroneous code. An Ada routine cannot ignore an exception.

3.2. Portability

The algorithms were first implemented in DEC Ada under VMS. They were then re-implemented in DEC C under VMS, then the Ada version was ported to Verdix Ada on a Sun under UNIX, and the C version ported to Sun C under UNIX. Both the Ada port and the C port had problems. After porting from DEC to Sun, the C code was ported back from Sun to DEC, with far fewer problems. The problems fell into two categories; operating system dependencies, and language implementation differences.

3.2.1. Operating system dependencies

The most basic problem was with the trig and square root math functions. Since these functions are not standard in either Ada or C, they had to be re-implemented to be ported. In each case, they were implemented by importing a run-time library function supplied by the operating system. VMS provides the functions MTH$SINCOSD (returns sin and cos of argument in degrees), MTH$ATAND2 (returns angle in degrees whose sin and cos are given), and MTH$SQRT, all in F_FLOAT (32 bit). UNIX provides _sin, _cos, _atan2, using radians, and _sqrt, all in 64 bit IEEE floating point. The VMS functions raise VMS conditions when they detect errors (such as both arguments zero for atan2). The UNIX functions set an error variable, but only if the 68881 co-processor error interrupt is enabled. These operating system differences cause different problems for Ada and C.

In Ada, the imported functions were hidden in a package body, and it was fairly straight-forward to edit that body. The exception handling is not so easy. The VMS conditions can be imported as Ada exceptions, via the DEC pragma IMPORT_EXCEPTION, which occurs in the package specification. The UNIX error variable is an external object, which must be imported via the Verdix pragma IMPORT_OBJECT, which also occurs in the package specification. Thus the specification of the math package changed, although in a minor way. In testing, it was discovered that the 68881 error interrupt is not normally enabled, so explicit checks that raised the corresponding exceptions were used instead of the UNIX error handling.

In Verdix 5.5, a machine code math package was written, allowing single precision functions. This package is not portable to other machines, but is appropriate for porting to 68881 systems without UNIX, although this portability was not tested.

In C, linking to the math functions was accomplished simply by calling the corresponding linker name. This name was easy to edit. Again, the exception handling caused more problems. Under VMS, the LIB$ESTABLISH routine was used to establish an exception handler for the math exceptions, in the inverse kinematics routine. Thus there is operating

system dependent code in the highest level routine. To avoid this, a different approach was used under UNIX; a convention of returning a status was adopted, similar to that in the C IO functions. This was a significant modification to the inverse kinematics routine. It should be noted, however, that when the resultant code was ported back to VMS, there were no problems in this area.

The test routines that verify the application code generate output files. Under VMS, each new file receives a new version number, and the old one is still there. Under UNIX, there is no equivalent mechanism. The first time I ran the test routines, I wiped out the files I had intended to compare the results of the test with. This was more of a nuisance than a real problem; I simply had to rename files before running the test routines.

A more serious problem had to do with the size of Ada file the system could compile. Especially when running suntools, the Sun system did not have as much ram available to the compiler as the microVAX. In several cases, this caused the Ada compiler to abort the compilation. Fortunately, it was easy to declare some function bodies to be separate, thus reducing the size of each compilation. The C compiler had no similar problems.

3.2.2. Language implementation differences

The differences in the Ada implementations were all minor. Some of the differences in the C implementations were more significant.

Both DEC and Verdix Ada had bugs in the TEXT_IO.ENUMERATION_IO package, when doing GET from a string. This function was used in reading data from the test files. The bugs were different, requiring different work-arounds (fortunately not mutually exclusive).

DEC requires a .ada file suffix, while Verdix requires a .a suffix. Another nuisance - since UNIX cannot rename a list of files, I had to write a short shell script.

The Verdix compiler encountered an internal error when compiling the Ada inverse_kinematics routine. Simplifying an expression by using a temporary variable cured the problem.

The syntax supported by the two C compilers is significantly different in some areas, particularly function declarations. DEC C allows declaring the type and name of each parameter in an external function declaration. This makes the declaration more readable, although it is not clear that the information is used by the compiler. Sun C does not support this syntax, so all the function definitions had to be edited. Since the syntax did not serve a real purpose in the DEC system, it was no real loss.

A more serious difference is in the initialization of aggregate automatic variables. DEC C supports such initialization, Sun does not, requiring a laborious sequence of assignments.

The Sun C compiler reported problems that the DEC C compiler did not; some variables were multiply defined, and an IO function used a single float format to read a double float.

In conclusion, both Ada and C are fairly portable. If care is taken to avoid operating system dependencies (using "plain vanilla" Ada or C, small enough files), both languages become much more portable.

3.3. Execution speed

Forward kinematics				
compiler	machine	millisec	relative speed, on each machine	
SMACRO	8086/8087	6.21	-	
DEC Ada	μVAX II	9.87	0.82	double precision
DEC Ada	μVAX II	8.06	1.00	reference version
DEC Ada	μVAX II	6.11	1.32	checks suppressed
DEC Ada	μVAX II	4.90	1.64	low-level routines
DEC C	μVAX II	4.68	1.72	
DEC Ada	μVAX II	3.69	2.18	checks suppressed, low-level routines
Verdix 5.41	Sun 3/160	12.60	1.00	reference version
Verdix 5.41	Sun 3/160	11.40	1.11	checks suppressed
Verdix 5.41	Sun 3/160	8.06	1.47	low-level routines
Verdix 5.41	Sun 3/160	7.81	1.61	checks suppressed, low-level routines
Verdix 5.5	Sun 3/160	9.75	0.80	double precision
Verdix 5.5	Sun 3/160	8.97	0.87	reference version (unix math)
Verdix 5.5	Sun 3/160	7.78	1.00	reference version (machine_code math)
Verdix 5.5	Sun 3/160	7.46	1.04	checks suppressed
Verdix 5.5	Sun 3/160	4.48	1.74	low-level routines
Verdix 5.5	Sun 3/160	4.36	1.78	checks suppressed, low-level routines
Sun C	Sun 3/160	3.41	2.28	

Figure 5a. Forward kinematics execution times.

Figure 5 gives the execution times for the various machine/compiler combinations. (Figure 5a shows forward kinematics, 5b shows inverse). The times are grouped by machine (except for Verdix 5.41), in order of increasing speed. The relative speed is the ratio with the reference Ada version on the same machine; thus DEC C is 72 percent faster at forward kinematics than DEC Ada, both running on a microVAX. Figure 6 gives an inter-machine comparison, using the fastest Ada version for each machine. (The various Ada versions are discussed below). Note that for Verdix 5.5, there are two versions of the math package; one importing the UNIX calls, and one in machine code. The machine code version was used for all tests except the one labeled "unix math".

The 8086/8087 runs at 8 MHz. The Sun is using a 68020 with a 68881, both at 16 MHz. SMACRO times are measured with hardware timer accurate to ~ 10 microseconds. DEC times are measured with VMS system timing routines, around a loop running 500 iterations. The microVAX was not busy at the time of measurement (only one user logged in). Sun times are measured using the UNIX /bin/time facility, around a loop running 10,000 iterations, on a non-busy machine (no suntools, only one user). There is no practical way to measure the overhead on the Sun and VAX, so these times are probably a bit long.

Inverse kinematics				
compiler	machine	millisec	relative speed, on each machine	
SMACRO	8086/8087	5.28	-	
DEC Ada	μVAX II	9.91	0.84	double precision
DEC Ada	μVAX II	8.28	1.00	reference version
DEC Ada	μVAX II	6.09	1.36	low-level routines
DEC Ada	μVAX II	6.04	1.37	checks suppressed
DEC Ada	μVAX II	4.59	1.80	checks suppressed, low-level routines
DEC C	μVAX II	4.20	1.97	
Verdix 5.41	Sun 3/160	10.50	1.00	reference version
Verdix 5.41	Sun 3/160	9.44	1.11	checks suppressed
Verdix 5.41	Sun 3/160	7.25	1.45	low-level routines
Verdix 5.41	Sun 3/160	6.53	1.61	checks suppressed, low-level routines
Verdix 5.5	Sun 3/160	6.35	0.79	double precision
Verdix 5.5	Sun 3/160	5.72	0.88	reference version (unix math)
Verdix 5.5	Sun 3/160	5.04	1.00	reference version (machine_code math)
Verdix 5.5	Sun 3/160	4.86	1.04	checks suppressed
Verdix 5.5	Sun 3/160	3.63	1.39	low-level routines
Verdix 5.5	Sun 3/160	3.51	1.44	checks suppressed, low-level routines
Sun C	Sun 3/160	3.20	1.58	

Figure 5b. Inverse kinematics execution times.

compiler	machine	millisec	relative speed	
Forward kinematics				
Verdix 5.41	Sun 3/160	7.81	0.80	checks suppressed, low-level routines
SMACRO	8086/8087	6.21	1.00	
DEC C	μVAX II	4.68	1.33	
Verdix 5.5	Sun 3/160	4.36	1.42	checks suppressed, low-level routines
DEC Ada	μVAX II	3.69	1.68	checks suppressed, low-level routines
Sun C	Sun 3/160	3.41	1.82	
Inverse kinematics				
Verdix 5.41	Sun 3/160	6.53	0.81	checks suppressed, low-level routines
SMACRO	8086/8087	5.28	1.00	
DEC Ada	μVAX II	4.59	1.15	checks suppressed, low-level routines
DEC C	μVAX II	4.20	1.26	
Verdix 5.5	Sun 3/160	3.51	1.50	checks suppressed, low-level routines
Sun C	Sun 3/160	3.20	1.65	

Figure 6. Inter-machine comparison

In order to evaluate the impact of various design decisions on execution speed, several versions of the Ada code were timed. The changes affected timing in different amounts between the two machines, and between the two procedures. The reference version uses the code presented above, with all run-time checks enabled, full compiler optimization, and single precision (32 bit) floating point. In the first modification, all run-time checks were suppressed. This gained about 30% on the VAX, and 4% to 10% on the Sun. Apparently the Verdix compiler is either better at eliminating redundant checks in the reference version, or the checks are faster.

For the second modification, the lowest level type declarations were changed to double precision. This lost about 17% on the VAX, and 15% on the Sun.

For the third modification, the code was made less abstract: the high-level multiply routines were bypassed, calling the low-level routines directly. The resulting code is closer to the C version (see figure 7). This gained about 40% for inverse kinematics, and 64% for forward kinematics, on both the VAX and the Sun. The reason this code is faster is that the high-level routines must determine at run-time what representation is used for the arguments. This involves executing a nested case statement, and calling the appropriate low-level routine. With checks left on, each call of a low-level routine also involves one or two discriminant checks. All of this overhead is bypassed when the low-level routines are called directly. Since the reference version of the forward kinematics was coded more abstractly than the inverse kinematics, it gained more speed from being made less abstract.

Finlly, the first and third modifications were combined, (checks suppressed on low-level code). This yielded better than a factor of two improvment in the DEC forward kinematics.

```
procedure Forward_Kinematics (
    Geometry : in GEOMETRY_RECORD;
    Joint    : in JOINT_ARRAY;
    Rot_Rep  : in Rotations.ROT_REPS;
    Pose     : in out Poses.POSE;
    Flags    : out CONFIG_FLAGS)
is
    -- Find pose and flags from joints.
    Trig_0 : TRIG_PAIR := Sin_Cos (Joint (0));

begin
    case Rot_Rep is
    when Rotations.MATRIX =>
         -- find matrix pose

    when others =>
         -- find quaternion pose
        declare
             use Rotations.Quaternions, Rotations.Angle_Frame_Axes,
                  Translations.Distance_Frame_Axes, Translations.Cartesians;

             Rot : QUATERNION := To_Quaternion
                  (Rotations.Trig_Frame_Axes.To_Trig_Axis ((Trig_0, Z)));

             Tran : CARTESIAN := Rotations.Trig_Frame_Axes."*" ((Trig_0, Z),
                  DISTANCE_FRAME_AXIS'(Geometry.Shoulder, Y));
        begin
             Rot  := Rot * ANGLE_FRAME_AXIS'(Joint (1), Y);
             Tran := Tran + Rot * DISTANCE_FRAME_AXIS'(Geometry.Upper_Arm, Z);

             Rot  := Rot * ANGLE_FRAME_AXIS'(Joint (2), Y);
             Tran := Tran + Rot * DISTANCE_FRAME_AXIS'(Geometry.Fore_Arm, Z);

             Rot  := Rot * ANGLE_FRAME_AXIS'(Joint (3), Z)
                  * ANGLE_FRAME_AXIS'(Joint (4), Y)
                  * ANGLE_FRAME_AXIS'(Joint (5), Z);

             Pose := ((Rotations.QUATERNION, Rot),
                  (Translations.CARTESIAN, Tran));
        end;
    end case;

    -- compute flags
end Forward_Kinematics;
```

Figure 7. low-level Ada code for forward kinematics

To gain a rough measure of how much overhead each language introduces, we estimate the optimal computation times for each machine, assuming no overhead for function calls, loading and storing operands, branching, etc. This will not give an accurate time estimate; the overhead for the operations left out is significant. However, the compiler is free (to some

extent) to optimize these overhead operations, but it must include the math operations. Also, the count of these operations is often used to give a measure of the complexity of an algorithm; it is interesting to see how well this corresponds with actual execution time. Figure 8 gives the number of clock cycles for each of the math operations for the 8087 and

operation	68881	8087
atan:	25	81
sin_cos :	28	71
sqrt :	7	23
div	6	25
mult :	4	17
add :	3	11
scale :	3	4

Figure 8. Execution times (micro-seconds) for various operations

68881 (times for the microVAX were unavailable). The number of operations for the forward kinematics is:

6 sin_cos, 1 sqrt , 76 mult, 43 add, 6 scale_by_2

and for the inverse kinematics:

6 atan2, 6 div, 4 sqrt, 45 mult, 26 add, 6 scale_by_2.

Figure 9 shows the estimated optimal times, and the percent utilization for each machine/compiler. The SMACRO code is essentially hand optimized assembler, which explains its high utilization. Verdix 5.41 achieves about half the utilization of Sun C, and Verdix 5.5 is almost as good as Sun C.A better measure of the efficiency of each compiler would be to compare each with an implementation by an expert in assembler. This was not

done, due to the amount of labor involved. Still, the current measure gives a fair ranking of compilers on a particular machine.

compiler	machine	actual	optimal	utilization
Forward kinematics				
Verdix 5.41	Sun 3/160	7.08	0.61	9 %
SMACRO	8086/8087	6.21	2.24	36 %
Verdix 5.5	Sun 3/160	4.36	0.61	14 %
Sun C	Sun 3/160	3.41	0.61	18 %
Inverse kinematics				
Verdix Ada	Sun 3/160	6.23	0.47	8 %
SMACRO	8086/8087	5.28	1.80	34 %
Verdix 5.5	Sun 3/160	3.51	0.47	11 %
Sun C	Sun 3/160	3.20	0.47	15 %

Figure 9. Actual and estimated optimal computation time (milliseconds), and percent utilization

4. Conclusions

The following sections discuss abstraction, portability, and speed. Recall that the task involved in the study is robot kinematics; this is typical of tasks in the lower levels of the NASREM hierarchy (PRIM and SERVO), but not of tasks in the higher levels. In particular, at the higher levels, it will be more appropriate to use more abstract code, because the tasks are more complex.

4.1. Abstraction

In summary, the Ada code is somewhat closer to the user's view of the problem. This was more true for the forward kinematics than the inverse, and I believe that it will be even more true for higher level functions such as trajectory planning and object recognition. If the impact of abstraction on execution time is a concern, abstractions should be used for the prototype code, during algorithm development. After the algorithm is proven, it can be speeded up by bypassing the high-level, abstract calls, and calling the low-level functions directly. In this way, low-level coding bugs are separated from high-level algorithm bugs, smoothing the debugging process.

The features of Ada that provide abstraction are aggregates and overloading, and to some extent strong typing and exception handling. Aggregates allow an abstract entity to be expressed as a whole, rather than always as individual elements. In addition, arguments to functions may be variables or aggregates, whereas in C two different functions must be provided, or explicit temporary variables must be used.

Overloading aids abstraction, since functions with similar meanings, but operating on different types, can have the same names. For example, all multiply routines can be named "*".

Strong typing is useful in debugging applications that use aggregates, particularly records with variant parts. Ada checks that the referenced variant is actually present, while C simply assumes it is. Once the code is proven, this check can be turned off to increase speed.

Exception handling lets the unusual events be handled in a separate section of code, leaving the main flow simpler.

4.2. Portability

The main conclusion to be drawn from the portability study is that it is easier to write portable code once it has been ported. The port of C code from Sun to VAX was much easier than the first port to Sun, simply because known problems were avoided in the first place.

Writing in Ada is no guarantee of portability; there were just as many problems (although of a different nature) with the Ada port as there were with the C port. Again, once the potential problems are known, it is easier to avoid them.

4.3. Timing

Currently, DEC Ada is faster than DEC C for forward kinematics, DEC C is faster for inverse kinematics, and Sun C is fastest overall. This assumes we allow Ada to use single precision. I believe this is justified: the SMACRO system has been running real robot applications for years using single precision. If C were to be modified to allow single precision, it would become the clear speed leader.

An unsuccessful attempt was made to upgrade C to single precision on the VAX; the DEC C compiler supports a single precision option. Unfortunately, this option does not go far enough: arguments to functions are still passed as double, and since almost everything gets passed to some function at some point, no real gain is made. In addition, mixing single and double proved to lead to lots of programming errors, which were very hard to debug. No similar attempt was made for Sun C.

Using abstract code does incur a speed penalty. For time-critical code, this penalty can be removed by first coding abstractly to prove the algorithm, then re-coding the time-critical parts less abstractly. Similarly, the run-time checks required by Ada can be left on during debugging, and turned off for real execution.

4.4. Overall summary

On the whole, Ada was judged to be the best language for this application. The degree of abstraction available made prototype debugging significantly easier, and the final speed was adequate, particularly on DEC hardware. The Verdix compiler (and others) can be expected to improve, as demonstrated by the significant improvement from Verdix 5.41 to 5.5.

5. References

[Booch 86] Grady Booch, Software engineering with Ada, Second Edition. Benjamin / Cummings, Menlo Park, California, 1986

[Craig 86] John J. Craig, *Introduction to Robotics: Mechanics and Control*, Addision-Wesley, Reading, Massachusetts 1986

[Hamilton 1844] Sir William Rowen Hamilton, "On Quaternions; or on a new system of imaginaries in algebra", *Philosophical Magazine* xxv pp 10-13 (July 1844).

[Kane 83] Thomas R. Kane, Peter W. Likins, David A. Levinson, *Spacecraft Dynamics*, McGraw-Hill Book Company, New York, 1983.

[NASREM 87] James S. Albus, Harry G. McCain, Ron Lumia, "NASA/NBS Standard Reference Model for Telerobot Control Sytem Architecture (NASREM)", NASA Document SS-GSFC-0027, December 4, 1986

[Paul 81] Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, Massachusetts, 1981.

[Purves 87] Lloyd Purves, "Telerobotic Services for the Space Station", Proceedings of the 1987 IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, 1987.